CLIPS-OWL: A Framework for Providing Object-Oriented Extensional Ontology Queries in A Production Rule Engine

G. Meditskos*,[a], N. Bassiliades[a]

[a]*Department of Informatics, Aristotle University of Thessaloniki, 54124, Greece*

Abstract

In this paper, we define a framework, namely CLIPS-OWL, for enabling the CLIPS production rule engine to represent the extensional results of DL reasoning on OWL ontologies in the form of Object-Oriented (OO) models. The purpose of this transformation is to allow CLIPS to use these OO models as static query models that are able to answer extensional ontology queries directly by the RETE reasoning engine during the development of custom CLIPS production rule programs, without interfacing at runtime the external DL reasoner. In that way, any CLIPS-based application may enhance its functionality by incorporating ontological knowledge without modifying the architecture of the CLIPS rule engine. CLIPS-OWL has been implemented using the Pellet DL reasoner and the CLIPS Object-Oriented Language (COOL).

Key words: production rules, ontologies, object-oriented model, CLIPS, OWL

1. Introduction

Data integration and transformation are issues that concern many users that wish to utilize Web data or to combine heterogeneous systems, since data are stored on the Web under different formats and each application can only handle a specific format [1][2][3]. To enable a specific tool to manipulate

*Corresponding author

Email addresses: `gmeditsk@csd.auth.gr` (G. Meditskos), `nbassili@csd.auth.gr` (N. Bassiliades) or just use data coming from various sources, a transformation phase must take place in order for the source information to be mapped to the format expected by the application.

The Semantic Web initiative[1] works on standards, technologies and tools in order to give to the information on the Web a well-defined meaning, enabling computers and people to work in better cooperation. Ontologies can be considered as a primary key towards this goal since they provide a controlled vocabulary of concepts, each with an explicitly defined and machine processable semantics [4]. The Web Ontology Language (OWL) [5][6] is the W3C recommendation for creating and sharing ontologies on the Web based on the Description Logic (DL) knowledge representation formalism [7]. It provides the means for ontology definition and specifies formal semantics that OWL ontology reasoners [8][9] use in order to derive new information, such as the OWL DL reasoners [10][11][12] that implement DL algorithms [13][14].

Rule-based systems have been extensively used in several applications and domains, such as e-commerce, personalization, businesses and academia. In the last years, the ability of manipulating and/or using semantically annotated information in rule systems has been considered as of great importance for both businesses and the successful proliferation of Semantic Web technologies, since it enables the already existing and well-known infrastructure of rule engines to gain access in the new evolution of Semantic Web.

In this paper, we present an approach that transforms the extensional knowledge that is derived from OWL DL reasoning into the OO model that is supported by the CLIPS production rule engine (COOL language) [15]. The idea of our framework, namely CLIPS-OWL, is to allow CLIPS to access this knowledge by querying the local OO KB that is generated after applying our transformation procedure on top of the DL reasoner and not by querying directly the external DL reasoning module, an architecture that would require considerable amount of changes in the initial architecture of CLIPS.

The motivation/contribution of our work, which is described in detail in section 2.5, can be summarized in the following:

- We enable the highly efficient and robust CLIPS production rule engine to import and query extensional OWL ontological knowledge, based on the inferencing capabilities of DL reasoners [10][11][12].

[1] `http://www.w3.org/2001/sw/`

- CLIPS-OWL can be easily embedded in already existing CLIPS-based environments since it is based on the native CLIPS capabilities without interfacing physically the DL reasoner.

It should be noted that CLIPS-OWL incorporates the extensional results of DL reasoning in CLIPS production rule programs based on the closed-world assumption (CWA), that is, only explicitly stated knowledge can be queried, and the unique name assumption (UNA), that is, all the objects of the OO model are different from each other [16].

CLIPS-OWL is suitable in application domains where ontologies are used as static models for sharing knowledge among heterogeneous environments. An example of such an application domain, in which CLIPS-OWL is already applied, is the domain of Software Antipatterns [17]. Antipatterns provide information on commonly occurring solutions to problems that generate negative consequences during software development. The antipatterns can be documented, stating how they are related with other antipatterns through special attributes, such as causes, symptoms and consequences. In order to enable the sharing of this knowledge, antipatterns can be documented in terms of an OWL ontology. However, the process of detecting which antipatterns exist in a software project is a challenging task which requires expert knowledge. Based on an antipattern ontology, CLIPS-OWL is used in order to enable a set of object-oriented CLIPS production rules to run and retrieve antipatterns relevant to some initial symptoms.

The rest of the paper is structured as follows: in section 2 we present the background and the motivation of our work. In section 3 we describe in detail the functions that transform the results of DL reasoning into the OO model of CLIPS. In section 4 we present an example of a rule-based application that uses the results of CLIPS-OWL. Finally, in sections 5 and 6, we present related work and we conclude, respectively.

2. Background and Motivation

2.1. The Web Ontology Language

The Web Ontology Language (OWL) is the W3C recommendation for defining and sharing ontologies in the Web and its theoretical background is based on the Description Logic (DL) [7] knowledge representation formalism, a subset of predicate logic. The modeling in OWL is performed using concepts, properties and instances. The first two constitute the terminological knowledge of the ontology (schema), whereas the latter constitutes the extensional knowledge [18].

The concepts in OWL represent sets of instances and by default, they are subsumed by the built-in concept `owl:Thing`. The set-oriented nature of OWL accounts for a great degree of expressiveness during concept definition, enabling the utilization of semantics relevant to sets, such as intersection.

The properties in OWL are classified in two categories: object properties and datatype properties. The former take as values instances, whereas the latter take data values, for example, integers, strings, etc. The properties are defined in terms of domain and range restrictions. The former denotes the concept(s) where instances should belong in order to use the property and the latter denotes the concept(s) or the datatype where the values should belong. Furthermore, a property maybe given special characteristic, such as symmetric, transitive, inverse and others.

The instances in OWL may belong to one or more concepts (instance concept membership) and may define more than one value in their properties. The instance concept membership set is computed after reasoning based on the subclass transitivity (class subsumption), as well as, based on the complex class constructors that are defined, such as intersection, union, equivalence and property restrictions.

Furthermore, two or more instances in OWL may be defined as identical (`owl:sameAs` property), although they have different IDs. This means that the instances have the same instance concept membership sets and the same values in their properties. It should be noted that OWL does not follow the unique name assumption (UNA) and therefore, instances with different IDs may refer to the same resource.

2.2. OWL DL reasoning

A DL reasoner is employed in order to infer any implicit relationship that stems from the asserted axioms of an ontology. At the level of the terminological knowledge, the reasoning procedure is called TBox reasoning, whereas at the extensional level is called ABox reasoning. Based on a set of ontology axioms, the basic reasoning procedures of a DL reasoner are:

- **Computation of the subsumption hierarchy.** This involves the computation of the subclass relationships among the concepts of the ontology. The reasoner also checks the ontology for inconsistencies.

- **Realization.** This is the computation of the instance concept membership sets, that is, the complete set of concepts where the instances belong and the instance property values.

2.3. The CLIPS Production Rule Engine and the COOL Language

CLIPS [15] is a RETE-based production rule engine written in C that was developed in 1985 by NASA's Johnson Space Center and it has undergone continual refinement and improvement ever since. One of the most interesting capabilities of CLIPS is that it integrates the production rule paradigm with the OO model, which can be defined using the COOL (CLIPS Object-Oriented Language) language. In that way, classes, attributes and objects can be matched on the production rule conditions (LHS), as well as to be altered on rules actions (RHS).

The semantics of CLIPS are the usual production rule semantics: rules whose condition is successfully matched against the current data are triggered and placed in the conflict set. The conflict resolution mechanism selects a single rule for firing its action, which may alter the data. Rule condition matching is performed incrementally, through the RETE algorithm.

2.3.1. COOL syntax and semantics

The COOL language of CLIPS provides the necessary constructs in order to define classes with attributes and the corresponding data model, that is, the objects and their attribute values.

Classes and attributes. The classes in COOL are regarded as types of objects and may define attributes. Usually, the classes of a domain are organized in terms of a class hierarchy using subclass relationships, where each class may have more than one direct superclasses (multiple direct inheritance). In that way, each class inherits also the attributes of all of its (direct or indirect) superclasses. Furthermore, subclass cycles are forbidden and therefore, two classes cannot share a mutual subclass relationship.

The attributes in COOL can take either a single value and they are called *slots*, or they can take more than one value and they are called *multislots*. For each attribute, a number of constraints may be defined, such as the valid type of the values that can take. The basic syntax for defining classes in COOL is shown below.

```
(defclass <name>
    (is-a <superclass-name>+)
    (<slot>* <constraint>*)
    (<multislot>* <constraint>*))
```

A class is defined by specifying the name, one or more superclasses[2], and zero, one or more attribute definitions with type constraints. We will be interested in the `type`, `allowed-classes`, `allowed-values` and `range` constraints only. The first one is used in order to restrict the type of data type attributes, such as strings, integers, etc. The second one denotes the class type of the objects that an attribute can take, using the `INSTANCE-NAME` value as a `type` constraint. The third one allows the attribute to be restricted to a specific set of values, for example {`yes no`}. Finally, the `range` constraint allows a numeric range to be specified for an attribute. As an example, we present the definition of the classes `Human` and `Man` with three properties `age`, `married` and `hasParent`. The former takes a single integer value, the second can take only the values yes or no and the latter takes object values of the class `Human`.

```
(defclass Human
    (is-a USER)
    (slot age (type INTEGER)(range 1 100))
    (slot married (type STRING)(allowed-values yes no))
    (multislot hasParent
        (type INSTANCE-NAME)(allowed-classes Human)))

(defclass Man
    (is-a Human))
```

Objects. An object in COOL is declared using a class as the direct class type. Based on the subclass hierarchy, the object inherits also the class types of the superclasses. In that way, the objects can use the attributes of their direct class types, as well as the attributes that are inherited from the superclasses.

An object is defined by specifying the name inside brackets, which is actually the object ID, the class type and any attribute value, using the following syntax:

[2]The built-in `USER` class of COOL must be the root class of the class hierarchy.

```
(make-instance [<name>] of <class>
    [(<attribute> <values>)]*)
```

For example, an object of the class `Man` above is defined as it follows:

```
(make-instance [george] of Man
    (age 28)
    (married no)
    (hasParent [Peter] [Mary]))
```

In that way, `[george]` is an object of the classes `Man` and `Human`.

2.4. Approaches for Using Ontologies with Rules

In this section, we briefly overview the approaches, both practical and theoretical, for the utilization of ontological knowledge in rule engines. More details can be found in the surveys [19], [20] and [21].

Interfacing external ontology reasoners with rule engines. In this scenario, the rule engine calls an external OWL reasoner, e.g. a DL reasoner, whenever is needed ("on the fly"). The hybrid approaches can be classified into *bidirectional* and *unidirectional*. In unidirectional frameworks, the information flows only from the ontology reasoner to the rule engine and thus, the ontology knowledge remains unmodified [22][23][24]. In bidirectional frameworks, the ontology predicates can be used both in the body and in the head of rules and thus, the ontology knowledge may be modified [25][26][27].

Mapping ontology reasoners on the data model of rule engines. In this case, the results of the external OWL reasoner are mapped on the data model that the rule engine supports, e.g. on triple-based facts [28][29]. In that way, the rule engine can operate without calling further the ontology reasoner (one-time mapping), since all the ontological knowledge exists in its KB.

Strong coupling of ontologies and rules. In this approach, also known as *homogeneous*, there is no external OWL reasoning module. The ontology semantics are partially mapped on a rule formalism [30][31], e.g. Datalog or *pd** entailment rules [32], that coexist in the rule base with rule predicates, enhancing the expressivity [33]. Therefore, a new reasoner is needed, able to handle the new homogeneous language that emerges [34][35][36][37][38].

2.5. CLIPS-OWL: The Basic Idea

CLIPS-OWL follows the approach of the one-time mapping of the results of DL reasoners. To give an example, consider the ontology in Table 1 that defines the class `Department`, the class `Chair` as subclass of the class `Professor` and the object property `isHeadOf` with domain and range restrictions the classes `Chair` and `Department`, respectively. Furthermore, there is the instance `csd` that belongs to the class `Department` and the instance `nick` of the class `Chair` with the property value `csd` in the property `isHeadOf`.

Table 1: Ontology example.

OWL Axioms (DL Syntax)
$\mathtt{Chair} \sqsubseteq \mathtt{Professor}$
$\top \sqsubseteq \forall \mathtt{isHeadOf}^{-}.\mathtt{Chair}$
$\top \sqsubseteq \forall \mathtt{isHeadOf}.\mathtt{Department}$
$\mathtt{csd} : \mathtt{Department}$
$\langle \mathtt{nick}, \mathtt{csd} \rangle : \mathtt{isHeadOf}$

Table 2: The extensional knowledge of the ontology in Table 1.

Instance	**Concept Membership**	**Property→Value**
csd	{Department}	
nick	{Chair, Professor}	isHeadOf→csd

By using a DL reasoner, it can be inferred that the instance `nick` belongs also to the concept `Professor`, due to the subclass relationship between the concepts `Chair` and `Professor`. The complete extensional knowledge of the ontology is depicted in Table 2. With CLIPS-OWL, our intention is to represent such extensional knowledge in terms of OO relationships in COOL. This involves the definition of an appropriate schema and data model in COOL, so that, for example, a query for the class types of the object `nick` should return both the classes `Chair` and `Professor` and a query for the `isHeadOf` attribute values of the object `nick` should return the object `csd`. Intuitively, the instance concept membership relationships should be represented in terms of object class type declarations and the instance property values should be represented as object attribute values. Below, we present the COOL model that CLIPS-OWL generates for the example.

```
(defclass Department (is-a owl:Thing))
(defclass Professor (is-a owl:Thing))
(defclass Chair (is-a Professor)
    (multislot isHeadOf
       (type INSTANCE-NAME) (allowed-classes Department)))
(make-instance [csd] of Department)
(make-instance [nick] of Chair (isHeadOf [csd]))
```

In the following, we elaborate on some of the main characteristics of CLIPS-OWL.

2.5.1. Powerful rule-based applications

The main functionality of CLIPS-OWL is to allow decision-making rule-based applications in CLIPS to run on top of ontological knowledge. It could be argued that such rule programs can be also developed using existing systems, such as RACER or Pellet that provide the infrastructure to run a set of rules on top of ontological knowledge. This argument is true, but it should be noted that the intention behind CLIPS-OWL is not to substitute or outperform existing implementations. CLIPS-OWL is an engine-specific framework, allowing CLIPS to be used as a production rule engine in order to develop rule-based applications on top of OWL ontologies. We consider this capability quite challenging, since CLIPS is a highly efficient general-purpose production rule engine with many years of development, having been widely used throughout the government, industry and academia. Therefore, we argue that it is a more powerful production rule engine with more capabilities compared to the rule engine implementations of, for example, RACER and Pellet that are mainly used for defining ontological relationships and not for building large and complex rule-based applications using traditional production rule engine features (conflict resolution strategies, message dispatching, salience, modules and many others).

2.5.2. Practical implementation

We have chosen to implement a mapping-oriented architecture in CLIPS for the following three reasons. Firstly, in a hybrid architecture, the rule engine is able to perform queries to the external reasoner only about the resources that exist in the KB, since they have to be matched by the rules. Therefore, the development of a rule program in a hybrid architecture requires the definition of a common vocabulary that would exist both in the rule engine and in the ontology. In mapping-oriented architectures, such as in CLIPS-OWL, the complete ontological knowledge is accessible and can be used by the rules, making more practical and straightforward the development of rule-based applications that use ontological knowledge.

Secondly, the implementation of a hybrid architecture using an existing general-purpose rule engine, such as CLIPS, is a very complex task compared to a rule engine that has been developed for this specific task. Any effort to embed the results of the external ontology reasoner calls directly into the RETE algorithm of CLIPS requires a great amount of changes to the initial infrastructure of CLIPS, resulting in a completely different architecture with unspecified side effects. Such an attempt might throw away decades of development on the efficient and robust RETE-based CLIPS rule engine.

Finally, the development of a protocol for the notification of the rule engine about potential changes to the ontological knowledge will have to be too complex, especially in a RETE-based production rule engine, since the RETE algorithm maintains intermediate caches in order to implement an incremental pattern-matching activity [19].

3. CLIPS-OWL Transformation Functions

Before defining the transformation procedure, it is important to clarify the way CLIPS-OWL handles two fundamental OWL characteristics that are not met in a typical OO environment and require special treatment: the concept equivalent semantics and the multiple instance concept memberships.

3.1. Concept Equivalence

A DL reasoner may infer that two or more ontology concepts are equivalent, having a mutual subclass relationship. At the extensional level, this means that the equivalent concepts have the same set of instances. This situation cannot be modeled in COOL, since subclass cycles are not allowed. To overcome this restriction, CLIPS-OWL follows an indirect way for preserving the concept equivalent semantics at the extensional level, by arbitrarily selecting a concept from the equivalent concept set as the *delegator* of each concept in the set. The idea is that (a) the delegator concept becomes subclass of all of its equivalent concepts in the COOL model, (b) the non-delegator concepts are defined as subclasses of `owl:Thing` and (c) every object class type declaration of the classes in the equivalent concept is transformed to a class type declaration only to delegators. In that way, all the equivalent classes share the same objects in COOL: the delegator class has all objects as direct objects, whereas the rest of the classes in the equivalent concept set have the delegator class as a direct subclass, therefore the delegator class' objects are their (indirect) objects, as well.

To exemplify, consider that a DL reasoner has inferred the three equivalent concepts C, D and E that have the same set of instances, let $\{i_1, i_2\}$. Assuming that CLIPS-OWL selects the concept C as the delegator of the equivalent concept set $\{C, D, E\}$, the class C would be defined in COOL as subclass of D and E. Furthermore, the corresponding objects of the instances i_1 and i_2 would be both defined in class C in COOL (the delegator). Therefore, a query in COOL for the objects of the classes C, D or E will return the same results, as the equivalent semantics of OWL impose for the corresponding concepts.

3.2. Multiple Instance Concept Membership

An instance in OWL may belong to multiple, hierarchically unrelated concepts at the same time. For example, an instance i may belong to the concepts C and D that do not have any subclass relationship. This also involves multiple domain restrictions of properties, where an instance that use such a property should belong to all the domain concepts. Taking into consideration that in an OO environment, such as in COOL, an object can be instantiated with a single class type, the direct transformation of each instance concept membership to an object class type declaration is impossible.

To overcome this limitation, the transformation procedure of CLIPS-OWL generates a system class for each multiple instance concept membership set that plays the role of the single class type that each object requires. The system classes are defined as subclasses of the most specific classes of the multiple instance concept membership set and the objects are instantiated with this system generated class type. To exemplify, a system class will be generated in CLIPS for the classes C and D, let T, and the object i will be instantiated with the class type T. In that way, a query to the common objects of the classes C and D will return the object i.

3.3. Transformation Procedure

The transformation procedure is defined on top of the reasoning results of the DL reasoner, which are the subsumption hierarchy, the domain/range property restrictions and the extensional knowledge. These results constitute the *input knowledge* of CLIPS-OWL.

Assuming that $TBox_R$ and $ABox_R$ are the TBox and ABox DL reasoning results, respectively, and that $COOL_{sch}$ and $COOL_{data}$ are the COOL OO schema and objects that are to be generated, respectively, the transformation is defined with two functions:

$$f_{sch} : TBox_R \rightarrow COOL_{sch}$$

$$f_{data} : ABox_R \rightarrow COOL_{data}$$

In the rest of the discussion, we represent the TBox results as a tuple of the form

$$TBox_R \equiv \langle CN_R, H_R, PN_R, DOM_R, OBJ_R, DAT_R \rangle,$$

where CN_R is the set with the named ontology concepts, PN_R is the set of the ontology properties, H_R is the set of the direct subclass relationships[3] between the concepts (represented with the symbol $\ll$), DOM_R is the set of property domain relationships , OBJ_R is the set of object property range relationships and DAT_R is the set of the datatype property range relationships. The sets H_R, DOM_R, OBJ_R and DAT_R are defined as follows.

$$H_R \equiv \{(x, y) \mid x \in CN_R \wedge y \in CN_R \wedge x \ll y\}$$

$$DOM_R \equiv \{(x, y) \mid x \in PN_R \wedge y \in CN_R \wedge \top \sqsubseteq \forall x.^{-}y\}$$

$$OBJ_R \equiv \{(x, y) \mid x \in PN_R \wedge y \in CN_R \wedge \top \sqsubseteq \forall x.y\}$$

$$DAT_R \equiv \{(x, y) \mid x \in PN_R \wedge y \in DAT_{RDF}{}^{4} \wedge \top \sqsubseteq \forall x.y\}$$

Regarding the ABox results, they are represented as a tuple of the form

$$ABox_R \equiv \langle IN_R, I_R, ObjV_R, DatV_R \rangle,$$

where IN_R is the set with the names (IDs) of the ontology individuals, I_R is the set with the direct instance class memberships, $ObjV_R$ is the set with the object property values of instances and $DatV_R$ is the set with the datatype values of instances. The sets I_R, $ObjV_R$ and $DatV_R$ are defined as follows.

$$I_R \equiv \{(x, y) \mid x \in IN_R \wedge y \in CN_R \wedge x : y\}$$

[3]The direct subclass relationship between two concepts x,y is defined as $x \ll y \equiv x \sqsubseteq y \wedge \nexists z(x \sqsubseteq z \wedge z \sqsubseteq y \wedge z \neq x \wedge z \neq y)$, where $\sqsubseteq$ is the DL symbol for concept subsumption.

[4]DAT_{RDF} are the allowed RDF Datatypes in OWL

$$ObjV_R \equiv \{(x,p,y) \mid x \in IN_R \wedge y \in IN_R \wedge \langle x,y\rangle : p \wedge (\exists c, (p,c) \in OBJ_R)\}$$
$$DatV_R \equiv \{(x,p,y) \mid x \in IN_R \wedge y \in VAL_{DAT}{}^{5} \wedge \langle x,y\rangle : p \wedge (\exists c, (p,c) \in DAT_R)\}.$$

The COOL schema is represented as a set of tuples of the form $\langle c, S^c, P^c\rangle$, where c is a class name, S^c is the set of the direct superclasses of c (represented using the notation $\ll_o$) and P^c is the set with the slot type definitions of c. More specifically,

$$COOL_{sch} \equiv \{\langle c, S^c, P^c\rangle \mid \forall s \in S^c, c \ll_o s \wedge \forall p \in P^c, p \in classSlots(c)\}$$

The COOL objects are represented as a set of tuples of the form $\langle i, c^i, PV_i\rangle$, where i is an object, c^i is the class type of i and PV^i is the set of the slot values tuples of i of the form $\langle p, v\rangle$. More specifically,

$$COOL_{data} \equiv \{\langle i, c^i, PV^i\rangle \mid i\ instanceOf\ c^i \wedge \forall \langle p,v\rangle \in PV^i, v \in i.p\}$$

The function f_{sch} transforms the TBox results into a COOL schema of class definitions that incorporate class slot declarations and slot types. The f_{data} transforms the ABox results into COOL objects with appropriate slot values. It should be noted that f_{sch} is an injective function, since each ontology concept is mapped on a unique COOL class. However, it is not a bijective function, since some COOL classes are generated that do not map any named ontology concept (non-surjective function). On the other hand, the f_{data} is a bijective function, since ontology instances are mapped on an one-to-one correspondence (see the B).

The transformation procedure involves two phases:

1. *Premapping phase.* The input to CLIPS-OWL is preprocessed in order to identify the delegators (section 3.3.1), to handle multiple domain and range restrictions (section 3.3.2), as well as, multiple instance class memberships (section 3.3.3).
2. *Mapping phase.* The results of premapping are mapped on the COOL model (section 3.3.4).

3.3.1. Delegator Management

The premapping phase starts by assigning delegators based on the function $f_{D_{sch}}$, which is defined as

$$f_{D_{sch}} : TBox_R \rightarrow TBox_D,$$

[5] VAL_{DAT} is the set with the allowed values for the DAT_{RDF} Datatypes

where $TBox_D \equiv \langle CN_D, H_D, PN_D, DOM_D, OBJ_D, DAT_D \rangle$. The purpose of this function is to determine the delegator of each named equivalent concept set and to define appropriate hierarchical relationships and domain/range property restrictions that refer only to delegators. The original sets of the concept names, properties and datatype property ranges are not modified, therefore, $CN_D = CN_R$, $PN_D = PN_R$ and $DAT_D = DAT_R$.

The delegators are defined in terms of the set DEL that contains pairs of the form (x, y), meaning that y is the delegator of x. More specifically,

$$\begin{aligned} DEL \equiv \{(x, y) \mid x \in CN_D \wedge \\ y = first(sort(\{c \mid (x, c) \in H_R \wedge (c, x) \in H_R\} \cup \{x\}))\}. \end{aligned}$$

The delegator concept y of x is the first element of the sorted collection[6] of all the equivalent concepts of x (concepts that have mutual subclass relationship with x). In that way, always the same concept is selected as the delegator for a specific equivalent concept set. If x does not have equivalent concepts, then it is the delegator of itself.

As we have described in section 3.1, the delegator becomes subclass of all of its equivalent concepts and the non-delegator concepts become subclasses of `owl:Thing` ($\top$). Furthermore, each subclass relationship should refer only to delegators. In that way, the H_D set is defined as

$$\begin{aligned} H_D \equiv \{(x, y) \mid (y, x) \in DEL\} \cup \{(y, \top) \mid (y, x) \in DEL\} \cup \{(x, y) | x \in CN_D \wedge \\ y \in CN_D \wedge (\exists a, (a, x) \in DEL) \wedge (\exists b, (b, y) \in DEL) \wedge (a, b) \in H_R\} \end{aligned}$$

Furthermore, the domain restrictions of properties and the range restrictions of object properties should refer now only to delegators. Therefore, the DOM_D and OBJ_D sets are defined as

$$\begin{aligned} DOM_D \equiv \{(x, y) \mid x \in PN_D \wedge y \in CN_D \wedge (\exists a, (a, y) \in DEL \wedge \\ (x, a) \in DOM_R)\} \end{aligned}$$

$$\begin{aligned} OBJ_D \equiv \{(x, y) \mid x \in PN_D \wedge y \in CN_D \wedge (\exists a, (a, y) \in DEL \wedge \\ (x, a) \in OBJ_R)\} \end{aligned}$$

[6]The collection is lexicographically sorted based on concept names.

3.3.2. Multiple Domain/Range Restrictions

An ontology property may be inferred to have multiple domain and/or range restrictions. In the case of multiple domain restrictions, an instance should belong to all the domain concepts in order to use the property. In the case of multiple range restrictions, the property value should belong to all the range concepts[7]. In order to handle these situations, the transformation procedure creates system generated concepts that play the role of the single property domains and ranges (see section 3.2).

The transformation function $f_{M_{sch}}$ takes as input the $TBox_D$ tuple and generates the $TBox_M$ tuple, that is,

$$f_{M_{sch}} : TBox_D \to TBox_M,$$

where $TBox_M \equiv \langle CN_M, H_M, PN_M, DOM_M, OBJ_M, DAT_M \rangle$. The sets PN_D and DAT_D do not participate in the transformation procedure and therefore, $PN_M = PN_D$ and $DAT_M = DAT_D$.

Initially, the set DOM_{MC} is computed that contains pairs of the form (x, MC), where x is a property and MC is the set with its domain(s):

$$DOM_{MC} \equiv \{(x, MC) \mid x \in PN_M \wedge MC \equiv \{c \mid (x, c) \in DOM_D\}\}.$$

Based on the DOM_{MC} set, the set DOM_M of $TBox_M$ is defined as the set that contains pairs of the form (x, y), where x is a property and y is its domain(s). In the case that x has multiple domains, a system concept is generated to play the role of the unique domain concept. More specifically,

$$\begin{aligned} DOM_M \equiv \{(x, y) \mid \exists y, (x, \{y\}) \in DOM_{MC}\} \cup \{(x, y) \mid \\ \exists MC, (x, MC) \in DOM_{MC} \wedge |MC| > 1 \wedge y = \mathit{classGen}(MC)\}. \end{aligned}$$

$\mathit{classGen}(C)$ is a function that generates a unique concept name that functionally depends on the set of concepts C. Notice that the name is not random, i.e. for the same set of concepts C always the same name will be generated. Furthermore, if concepts that are hierarchically related exist in C, only the most specific ones remain[8]. It should be noted that

[7]It is assumed that a datatype property can have only a single datatype restriction.

[8]After that elimination, a possible function definition could be $\mathit{classGen}(C) = \mathit{return}(\mathit{concatenate}(\mathit{sort}(C)))$.

$\forall (x,y) \in DOM_M, \nexists (x,z), y \neq z$, that is, all properties have single domain restrictions in DOM_M.

The same rational holds for the OBJ_M set, which is based on the set OBJ_{MC} that contains pairs of the form (x, MC), where x is a property and MC is the set with its range(s).

$$OBJ_{MC} \equiv \{(x, MC) \mid x \in PN_M \wedge MC \equiv \{c \mid (x,c) \in OBJ_D\}\}.$$

$$\begin{aligned} OBJ_M \equiv \{(x,y) \mid \exists y, (x, \{y\}) \in OBJ_{MC}\} \cup \{(x,y) \mid \\ \exists MC, (x, MC) \in OBJ_{MC} \wedge |MC| > 1 \wedge y = \mathit{classGen}(MC)\}. \end{aligned}$$

It should be noted that $\forall (x,y) \in OBJ_M, \nexists (x,z), y \neq z$, that is, all properties have single range restrictions in OBJ_M.

Due to the system generated concepts, the sets CN_M and H_M should contain the new concepts and the new hierarchical relationships, respectively. The set CN_M contains the new concepts that are generated due to the multiple domain and range restrictions:

$$CN_M \equiv CN_D \cup \{c \mid \exists x, (x,c) \in DOM_M\} \cup \{c \mid \exists x, (x,c) \in OBJ_M\},$$

The set H_M defines the hierarchical relationships between the system generated concepts and their superclasses, that is, the system generated concepts are subclasses of the corresponding concepts for which they are generated:

$$\begin{aligned} H_M \equiv & H_D \cup \{(x,y) \mid x \in CN_M \setminus CN_D \wedge y \in CN_D \wedge \\ & \exists w, \exists MC, (w, MC) \in DOM_{MC} \cup OBJ_{MC} \wedge |MC| > 1 \wedge \\ & x = \mathit{classGen}(MC) \wedge y \in MC\} \end{aligned}$$

3.3.3. Instance Management

The instance concept memberships should refer only to delegator concepts. The function $f_{D_{data}}$ is responsible for this task, where

$$f_{D_{data}} : ABox_R \rightarrow ABox_D,$$

generating a tuple $ABox_D \equiv \langle IN_D, I_D, ObjV_D, DatV_D \rangle$, where $IN_D = IN_R$, $ObjV_D = ObjV_R$, $DatV_D = DatV_R$. This function ensures that each reference to a non-delegator concept in I_R is substituted with its delegator, that is

$$I_D \equiv \{(x,y) \mid x \in IN_D \wedge y \in CN_D \wedge (\exists a, (a,y) \in DEL \wedge (x,a) \in I_R)\}.$$

In order to handle multiple instance concept memberships, the transformation procedure is based on the function $f_{M_{data}}$ that is defined as

$$f_{M_{data}} : ABox_D \rightarrow ABox_M.$$

This function generates a tuple $ABox_M \equiv \langle IN_M, I_M, ObjV_M, DatV_M \rangle$, where $IN_M = IN_D$, $ObjV_M = ObjV_D$ and $DatV_M = DatV_D$. The set I_M is determined by substituting each multiple instance concept membership with a single system generated concept, following a procedure similar to section 3.3.2. More specifically, the set I_{MC} is determined that contains pairs of the form (x, MC), where x is an instance and MC is the set of its concept memberships.

$$I_{MC} \equiv \{(x, MC) \mid x \in IN_M \wedge MC \equiv \{c \mid (x, c) \in I_D\}\}.$$

Then, the set I_M is computed that contains pairs of the form (x, y), where x is an instance and y is its concept membership. In case the instance have more than one concept memberships, that is, $|MC| > 1$, then a system concept is generated to play the role of the single concept membership.

$$\begin{aligned} I_M \equiv \{(x, y) \mid \exists y, (x, \{y\}) \in I_{MC}\} \cup \{(x, y) \mid \exists MC, (x, MX) \in I_{MC} \wedge \\ |MC| > 1 \wedge y = classGen(MC)\} \end{aligned}$$

It should be noted that $\forall (x, y) \in I_M, \nexists (x, z), y \neq z$, that is, all instances have a single concept membership in I_M. Furthermore, the function $f_{M_{data}}$ may generate new system concepts in order to handle multiple instance concept memberships. For that reason, in practice, both the CN_M and H_M sets are updated in order to incorporate also the new concepts.

3.3.4. Mapping Phase

The mapping of the $TBox_M$ tuple on the $COOL_{sch}$ set is performed by the function $f_{COOL_{sch}}$, which is defined as

$$f_{COOL_{sch}} : TBox_M \rightarrow COOL_{sch}.$$

$COOL_{sch}$ is a set that contains tuples of the form $\langle c, S^c, P^c \rangle$ that are presented in section 3.3 and it is defined as

$$\begin{aligned} COOL_{sch} \equiv \{\langle c, S^c, P^c \rangle \mid c \in CN_M \wedge S^c = \{s | (c, s) \in H_M\} \wedge \\ P^c = \{\langle p, t \rangle \mid (p, c) \in DOM_M \wedge (p, t) \in OBJ_M \cup DAT_M\}\}. \end{aligned}$$

More specifically, every class c in COOL exists in the set CN_M, the superclasses of c are determined based on the set H_M and the slot type definitions of c are determined based on the DOM_M, OBJ_M and DAT_M sets.

The mapping of the $ABox_M$ tuple on the $COOL_{data}$ set is performed by the function $f_{COOL_{data}}$, which is defined as

$$f_{COOL_{data}} : ABox_M \rightarrow COOL_{data}.$$

$COOL_{data}$ is a set that contains tuples of the form $\langle i, c^i, PV^i \rangle$ that are presented in section 3.3 and it is defined as

$$\begin{aligned} COOL_{data} \equiv \{\langle i, c^i, PV^i \rangle \mid i \in IN_M \wedge (i, c^i) \in I_M \wedge \\ PV^i = \{\langle p, v \rangle \mid (i, p, v) \in ObjV_M \cup DatV_M\}\} \end{aligned}$$

More specifically, every instance i in COOL exists in the set IN_M, the class type of i is determined based on the set I_M and the property values of i are determined based on the sets $ObjV_M$ and $DatV_M$.

Based on the transformation functions that are presented in this section, the overall procedures for generating the COOL schema and objects consist of two function compositions of the form

$$COOL_{sch} = f_{sch}(TBox_R) = f_{COOL_{sch}}(f_{M_{sch}}(f_{D_{sch}}(TBox_R)))$$

$$COOL_{data} = f_{data}(ABox_R) = f_{COOL_{data}}(f_{M_{data}}(f_{D_{data}}(ABox_R)))$$

Generating the COOL code. The generation of the actual COOL code based on the $COOL_{sch}$ and $COOL_{data}$ sets is straightforward. For each tuple $\langle c, S^c, P^c \rangle \in COOL_{sch}$, a `defclass` construct is generated with name c, `is-a` constraint the set S^c, and for each $\langle p, t \rangle \in P^c$, a `multislot` declaration is defined. The *dtMap* function maps an OWL datatype on a COOL datatype, according to Table 3. For each tuple $\langle i, c^i, PV^i \rangle \in COOL_{data}$, a `make-instance` construct is defined with name $[i]$, `of` constraint the class c^i, and for each $\langle p, v \rangle \in PV^i$, the value v is inserted into the slot p (Procedure 1)[9].

[9]The *to_instances* function is used in order to insert square brackets (`[,]`) in each value of the set, as the COOL object notation requires.

Procedure 1: *codeGenerator*$(COOL_{sch}, COOL_{data})$

```
foreach $\langle c, S^c, P^c \rangle \in COOL_{sch}$ do
    if $S^c =$ owl:Thing then
        $write($(defclass $c$ is-a USER)
    else $write($(defclass $c$ is-a $S^c$)
    foreach $\langle p, t \rangle \in P^c$ do
        if $objectProperty(p)$ then
            $write($(multislot $p$ (type INSTANCE-NAME)
            $write($(allowed-classes $t$)))
        else $write($(multislot $p$  $dtMap(t)$))
    $write($))
foreach $\langle i, c^i, PV^i \rangle \in COOL_{data}$ do
    $write($(make-instance [$i$] of $c^i$)
    $PVS^i = \varnothing$
    foreach $\langle p, v \rangle \in PV^i$ do
        if $\langle p, V \rangle \in PVS^i$ then
            $PVS^i = PVS^i \setminus \langle p, V \rangle \cup \langle p, V \cup \{v\} \rangle$
        else
            $PVS^i = PVS^i \cup \langle p, \{v\} \rangle$
    foreach $\langle p, V \rangle \in PVS^i$ do
        $write($($p$ $to\_instances(V)$))
    $write($))
```

Table 3: Basic mappings of OWL datatypes on COOL slot constraints

OWL datatype	**COOL data type**
`xsd:int`	`(type INTEGER)`
`xsd:float`	`(type FLOAT)`
`xsd:short`	`(type INTEGER) (range -32768 32767)`
`xsd:boolean`	`(type SYMBOL) (allowed-values true false)`
`xsd:string`	`(type STRING)`
`xsd:nonNegativeInteger`	`(type INTEGER) (range 0 ?VARIABLE)`
`xsd:integer`	`(type INTEGER)`
`xsd:anyURI`	`(type STRING)`
`xsd:positiveInteger`	`(type INTEGER) (range 1 ?VARIABLE)`
`xsd:nonPositiveInteger`	`(type INTEGER) (range ?VARIABLE 0)`

Table 4: Ontology example with university regulations.

#	OWL Axioms (DL Syntax)
$u1$	`FullProfessor` $\sqsubseteq$ `FacultyMember`
$u2$	`NonTeachingFullProfessor` $\equiv$ `FullProfessor` $\sqcap$ $\neg$`teaches.Course`
$u3$	`AdvancedCourse` $\sqcup$ `BasicCourse` $\equiv$ `Course`
$u4$	`AdvancedCourse` $\sqcap$ `BasicCourse` $\equiv \bot$
$u5$	`mary` : `FullProfessor` $\sqcap$ $\forall$`teaches.AdvancedCourse`
$u6$	`paul` : `Student`
$u7$	`john` : `FullProfessor`
$u8$	`ai` : `AdvancedCourse`
$u9$	`kr` : `Topic`
$u10$	`lp` : `Topic`
$u11$	$\langle$`john`, `ai`$\rangle$: `teaches`

4. Using CLIPS-OWL for Developing a Production Rule Program

Let the ontology of Table 4 that models the university domain, describing courses, professors and students. The goal is to develop a production rule program in CLIPS that derives facts of the form `(mayDoThesis x y)` that denote with which professor (`y`) a student (`x`) is eligible to do a thesis, based on the knowledge that is modeled in the ontology. Therefore, we use the ontological knowledge as the basis for developing a rule-based decision-making application in CLIPS. The ontology axioms model the following relationships:

- $u1$: A `FullProfessor` is also a `FacultyMember`
- $u2$: A `NonTeachingFullPRofessor` is a `FullProfessor` that does not teaches a course
- $u3$: The instance set of the concept `Course` is the union of the instances of the concepts `AdvancedCourse` and `BasicCourse`
- $u4$: The concepts `AdvancedCourse` and `BasicCourse` are disjoint
- $u5$: `mary` is a `FullProfessor` and teaches only advanced courses.
- $u6$: `paul` is a `Student`
- $u7$: `john` is a `FullProfessor`
- $u8$: `ai` (artificial intelligence) is an `AdvancedCourse`
- $u9$: `kr` (knowledge representation) is a `Topic`
- $u10$: `lp` (logic programming) is a `Topic`
- $u11$: `john` teaches `ai`

4.1. Transforming the Ontology Example

By applying a DL reasoner on the ontology example, we have for the $TBox_R$ results that:

$$\begin{aligned} CN_R \equiv \{ & \texttt{owl:Thing}, \texttt{FullProfessor}, \texttt{FacultyMember}, \texttt{AdvancedCourse}, \\ & \texttt{NonTeachingFullProfessor}, \texttt{BasicCourse}, \texttt{Course}, \texttt{Student}, \texttt{Topic}\} \end{aligned}$$

$$\begin{aligned} H_R \equiv \{ & (\texttt{owl:Thing}, \texttt{owl:Thing}), (\texttt{FullProfessor}, \texttt{FacultyMember}), \\ & (\texttt{NonTeachingFullProfessor}, \texttt{FullProfessor}), \\ & (\texttt{AdvancedCourse}, \texttt{Course}), (\texttt{BasicCourse}, \texttt{Course}), \\ & (\texttt{FacultyMember}, \texttt{owl:Thing}), (\texttt{Student}, \texttt{owl:Thing}), \\ & (\texttt{Topic}, \texttt{owl:Thing}), (\texttt{Course}, \texttt{owl:Thing})\} \end{aligned}$$

$$DOM_R \equiv OBJ_R \equiv \{(\texttt{teaches}, \texttt{owl:Thing})\}, DAT_R = \varnothing.$$

For the $ABox_R$ results, we have that

$$IN_R \equiv \{\texttt{mary}, \texttt{paul}, \texttt{john}, \texttt{ai}, \texttt{kr}, \texttt{lp}\}$$

$$\begin{aligned} I_R \equiv \{ & (\texttt{mary}, \texttt{FullProfessor}), (\texttt{paul}, \texttt{Student}), (\texttt{john}, \texttt{FullProfessor}), \\ & (\texttt{ai}, \texttt{AdvancedCourse}), (\texttt{kr}, \texttt{Topic}), (\texttt{lp}, \texttt{Topic})\} \end{aligned}$$

$$ObjV_R = \{(\texttt{john}, \texttt{teaches}, \texttt{ai})\}, DatV_R = \varnothing.$$

The ontology example does not contain any named equivalent concept set, nor multiple domain, range or instance class membership relationships. Therefore, $TBox_R = TBox_M$ and $ABox_R = ABox_M$. In that way, the $COOL_{sch}$ set is finally defined as

$$\begin{aligned} COOL_{sch} \equiv \{ & \langle \texttt{owl:Thing}, \varnothing, \{\langle \texttt{teaches}, \texttt{owl:Thing}\rangle\}\rangle, \\ & \langle \texttt{FullProfessor}, \{\texttt{FacultyMember}\}, \varnothing\rangle, \\ & \langle \texttt{AdvancedCourse}, \{\texttt{Course}\}, \varnothing\rangle, \\ & \langle \texttt{NonTeachingFullProfessor}, \{\texttt{FullProfessor}\}, \varnothing\rangle, \\ & \langle \texttt{BasicCourse}, \{\texttt{Course}\}, \varnothing\rangle, \\ & \langle \texttt{FacultyMember}, \{\texttt{owl:Thing}\}, \varnothing\rangle, \\ & \langle \texttt{Course}, \{\texttt{owl:Thing}\}, \varnothing\rangle \\ & \langle \texttt{Student}, \{\texttt{owl:Thing}\}, \varnothing\rangle \\ & \langle \texttt{Topic}, \{\texttt{owl:Thing}\}, \varnothing\rangle\} \end{aligned}$$

The $COOL_{data}$ set is defined as

$$\begin{aligned} COOL_{data} \equiv \{ & \langle \texttt{mary}, \texttt{FullProfessor}, \varnothing \rangle, \langle \texttt{paul}, \texttt{Student}, \varnothing \rangle, \\ & \langle \texttt{john}, \texttt{FullProfessor}, \{ \langle \texttt{teaches}, \texttt{ai} \rangle \} \rangle, \\ & \langle \texttt{ai}, \texttt{AdvancedCourse}, \varnothing \rangle, \langle \texttt{kr}, \texttt{Topic}, \varnothing \rangle, \langle \texttt{lp}, \texttt{Topic}, \varnothing \rangle \} \end{aligned}$$

By applying the code generator procedure using the $COOL_{sch}$ and $COOL_{data}$ sets, we have the following COOL code:

```
(defclass owl:Thing (is-a USER)
  (multislot teaches
     (type INSTANCE-NAME)(allowed-classes owl:Thing)))
(defclass FacultyMember (is-a owl:Thing))
(defclass Course (is-a owl:Thing))
(defclass AdvancedCourse (is-a Course))
(defclass NonTeachingFullProfessor (is-a owl:Thing))
(defclass FullProfessor (is-a FacultyMember))
(defclass Topic (is-a owl:Thing))
(defclass Student (is-a owl:Thing))
(defclass BasicCourse (is-a Course))
(make-instance [ai] of AdvancedCourse)
(make-instance [lp] of Topic)
(make-instance [john] of FullProfessor (teaches [ai]))
(make-instance [paul] of Student)
(make-instance [kr] of Topic)
(make-instance [mary] of FullProfessor)
```

Before describing the rule program, we give a short overview of the way the objects and facts can be matched in the condition of CLIPS production rules.

4.2. CLIPS Production Rules

A production rule in CLIPS is defined using the `defrule` construct that consists of conditions and actions separated with the symbol `=>`. The conditions can match both facts and objects, whereas the actions define the actions that should be taken upon the satisfaction of all the conditions. The facts consist of a symbol followed by a sequence of zero or more fields separated by spaces and delimited by an opening parenthesis on the left and a closing parenthesis on the right. The first field of an ordered fact specifies a relation that applied to the remaining fields in the ordered fact, for example, `(father-of jack bill)`. An example of a fact-based CLIPS rule is presented below.

```
(defrule test-rule1
    (refrigerator light on)
    (refrigerator door open)
=>
    (assert (refrigerator food spoiled)))
```

Objects of user-defined classes in COOL can be pattern-matched on the left-hand side of rules using object patterns of the form

```
<object-pattern> ::= (object <attribute-constraint>*)
<attribute-constraint> ::=  (is-a <constraint>) |
                            (name <constraint>) |
                            (<slot-name> <constraint>*)
```

The `is-a` constraint is used for specifying class constraints and it also encompasses subclasses of the matching classes. The `name` constraint is used for specifying a specific object on which to pattern-match. Constraints are also used in slots/multislots in order to restrict certain type of values. Both in fact and object patterns, it is possible to use variables in order to be matched with certain values. A single-value variable is denoted as `?x`, whereas a multi-value variable is denoted as `$?x`. An example rule that prints all the objects of the class `Person` is presented below.

```
(defrule test-rule2
    (object (is-a Person)
            (name ?x))
=>
    (printout t ?x crlf))
```

4.3. Example Rule Program

For the example rule program, we use the following facts and rules in CLIPS:

`(mayDoThesis ?x ?y)` “Student `?x` can do a thesis with professor `?y`”
`(curr ?x ?y)` “Student `?x` has topic `?y` in his/her curriculum”

`(expert ?x ?y)` "Professor `?x` is an expert on topic `?y`"
`(exam ?x ?y)` "Student `?x` passed the exam on topic `?y`"
`(subject ?x ?y)` "Course `?x` covers topic `?y`"

```
(defrule mayDoThesis
   (object (is-a Student)(name ?x))
   (object (is-a Topic)(name ?z))
   (object (is-a FacultyMember)(name ?y)
      (teaches $? ?c $?))
   (object (is-a AdvancedCourse)(name ?c))
   (curr ?x ?z)
   (expert ?y ?z)
=>
   (assert (mayDoThesis ?x ?y)))

(defrule curr
   (object (is-a Student)(name ?x))
   (object (is-a Topic)(name ?z))
   (object (is-a Course)(name ?y))
   (exam ?x ?y)
   (subject ?y ?z)
=>
   (assert (curr ?x ?z)))
```

More specifically, the `curr` rule asserts a fact `(curr ?x ?z)` if `?x` passed the exam in course `?y` that covers topic `?z`. The `mayDoThesis` rule asserts a fact `(mayDoThesis ?x ?y)` if `?x` has topic `?z` in the curriculum, `?y` is an expert on `?z`, and `?y` is a faculty member that teaches at least one advanced course. By loading the facts

```
(subject [ai] [kr]),
(subject [ai] [lp]),
(expert [mary] [lp]),
(expert [john] [kr]),
(exam [paul] [ai]),
```

and running the production rules, we result in the addition of the fact `(mayDoThesis [paul] [john])` in the KB. More specifically, the `curr` rule is activated and asserts the facts `(curr [paul] [kr])` and `(curr [paul]` `[lp])`. Then, the `mayDoThesis` rule is activated, since fact `(expert [john] [kr])` exists and fact `(curr [paul] [kr])` has been added by the `curr` rule. Note the way object patterns are used in order to restrict the objects to belong to certain classes or to have certain attribute values. For example, the `maydothesis` rule matches objects of the class `FacultyMember` that teach at least one advanced course `?c`, using multifield wildcards that produce every possible match combination.

4.4. Memory Consumption and Rule Activation Time

In order to test the efficiency of the OO representation against the trivial fact-based representation of the DL reasoner's semantics, we generated the COOL model and the set of the plain CLIPS facts of the UOBM DL ontology [39] (20 departments, UOBM-1) after Pellet DL reasoning. The COOL model was generated by the CLIPS-OWL library, whereas the CLIPS facts were extracted directly from Pellet. Table 5 depicts the memory requirements of CLIPS for loading each model, observing that CLIPS requires considerably less memory to load the OO model than the corresponding fact-based model.

Furthermore, the benchmark defines 13 extensional queries, that is, queries about instances. We expressed these queries in terms of object and fact pattern constraints in CLIPS production rules and we used these rules in order to test the rule activation performance, i.e. to test the query response capabilities of each data model. For example, the query 11 that retrieves all the objects of the class `Person` who `like` at least one similar thing to the head of the `University0`, is defined in the fact and in the COOL models as:

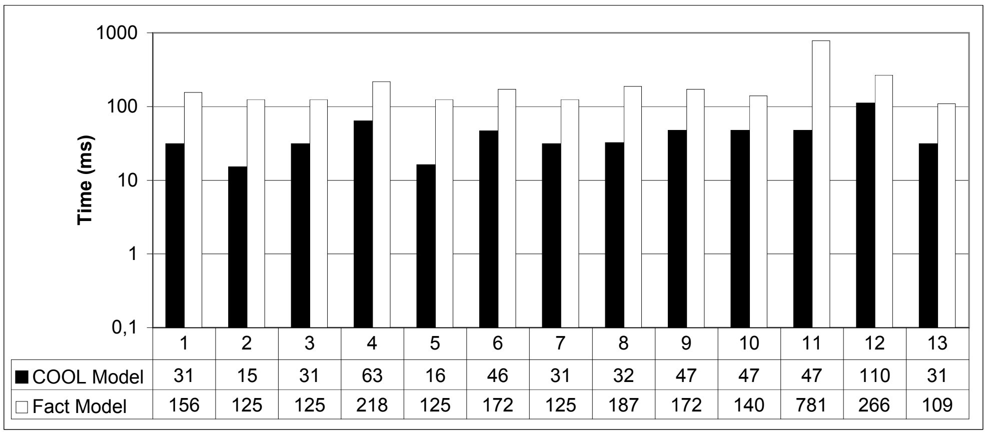
```
(defrule query11-fact
  (triple ?x rdf:type Person)
  (triple ?x like ?y)
  (triple ?z rdf:type Chair)
  (triple ?z isHeadOf University0)
  (triple ?z like ?y))
=>
  (printout t ?x crlf))

(defrule query11-cool
  (object
    (is-a Person)
    (name ?x)
    (like $? ?y $?))
  (object
    (is-a Chair)
    (name ?z)
    (isHeadOf $? [University0] $?)
    (like $? ?y $?))
=>
  (printout t ?x crlf))
```

Figure 1 depicts the rule activation time of each rule in the two models. In all queries, the OO constraints were checked faster that the corresponding fact-based, showing the efficiency of the OO representation that encapsulates all the related knowledge about a resource in a single object definition.

Table 5: Memory requirements and the number of classes, objects and facts.

	COOL Model	**Fact Model**
CLIPS memory	29 MB	75 MB
Number of classes	646	-
Number of objects	25,460	-
Number of facts	-	517,490



	1	2	3	4	5	6	7	8	9	10	11	12	13
■ COOL Model	31	15	31	63	16	46	31	32	47	47	47	110	31
□ Fact Model	156	125	125	218	125	172	125	187	172	140	781	266	109

Figure 1: Rule activation times.

5. Related Work

The similarities between OWL and OO modeling have been considered in many works. OntoJava [40] converts RDF Schema and RuleML into Java classes: every RDF class is mapped on a Java class, every RDF property on a class attribute and every RDF instance on a Java object. However, this mapping is not based on an RDF reasoner, but it tries to capture semantics using OO principles. Therefore, OntoJava is not able to perform, for example, sophisticated instance classification based on complex class expressions. In CLIPS-OWL, a richer mapping of OWL semantics on the COOL model is defined using a DL reasoner for handling the ontology semantics.

In [41], an approach is presented for mapping OWL on Java. Like OntoJava, it does not use a DL reasoner to infer the semantic relationships, but it uses directly OO principles. To this end, some OWL semantics are not handled, such as the inverse functional properties. Furthermore, the approach does not define a mapping of OWL instances on Java objects, but it deals only with concepts and roles. There are other similar approaches, such as the RDFReactor [42]. The major difference of such approaches to ours is that their goal is to enable the use of the ontological knowledge in OO programming, for example in Java, without being interested in complete and sophisticated OWL reasoning. The OO principles are not enough for complete ontology reasoning and a dedicated ontology reasoner should be used, as CLIPS-OWL does with the use of a DL reasoner.

The authors in [43] take advantage of a mapping of $\mathcal{ALC}^-$ ($\mathcal{ALC}$ without intersection, allowing complex concepts built with at most one construct of $\mathcal{ALC}$) on UML for performing class consistency. Furthermore, a mapping of UML on $\mathcal{DLR}_{ifd}$ and $\mathcal{ALCQI}$ is presented for allowing reasoning tools developed for DLs to reason on UML class diagrams. In our work, the generated COOL model is not used for consistency checking, but it serves as a query model for developing rule-based applications in CLIPS. The work in [43] has quite interesting potentials, however it has different motivations and application scenarios compared to ours.

In [44] a hybrid framework is presented where part of the domain exists in OWL and the other part in an OO model. This approach does not actually transform OWL into the OO model, but it rather defines an interface through which the two models can cooperate. This involves the manual separation of the domain into two KBs, based on some criteria that are presented in the paper. A major difference of this approach compared to ours is the hybrid nature. In section 2.4.3 we explain why such a hybrid architecture is very difficult to be implemented/embedded in an already existing RETE-based environment, such as CLIPS.

ActiveRDF [45] is an RDF mapping approach to the OO model in Ruby. The purpose of this mapping is to generate an API in order to manipulate and query RDF ontologies. However, the high expressivity of OWL requires a more sophisticated mapping than RDF, that we achieve by utilizing an OWL DL reasoner. We are not interested in the dynamic modification of the generated OO model, since we use the COOL model only as query infrastructure in CLIPS and not as an API.

SWCLOS [46] is an OWL reasoner developed on top of the Common Lisp Object System (CLOS) that allows LISP programmers to develop OO systems. SWCLOS maps ontologies on the OO model of CLOS and performs inferencing using entailment rules [32] and not a DL reasoner. Therefore, it is a homogenous approach, targeting mainly at (partial) ontology reasoning, in contrast to CLIPS-OWL that defines a mapping to the COOL language.

To the best of our knowledge, the only effort so far that allows CLIPS to use OWL knowledge is realized in O-DEVICE [47]. It is actually an OWL reasoner using the OO capabilities of CLIPS. The system applies entailment rules for OWL reasoning, and thus, it is a homogeneous system targeting mainly at OWL reasoning, just like SWCLOS. Although O-DEVICE is defined on top of CLIPS, the mapping and the generated OO model are different from the corresponding ones of CLIPS-OWL. This happens since the OO model in O-DEVICE is created taking into consideration entailment rules for OWL reasoning, whereas in CLIPS-OWL, the OWL semantics are derived directly from the DL reasoner. For example, O-DEVICE requires the definition of meta-objects, which are special objects necessary to store concept and role information. For that reason, O-DEVICE loads RDF and OWL meta-models. This approach is also followed by SWCLOS. In CLIPS-OWL there are no meta-models and meta-objects, resulting in a simpler COOL model. Also, CLIPS-OWL maps a richer set of OWL semantics than O-DEVICE, since it uses a DL reasoner for OWL reasoning instead of applying a limited OWL entailment rule set, such as O-DEVICE follows.

There is also an effort to map OWL ontologies on the Business Object Model (BOM) in JRules BRMS [19], using an intermediate Jena model. This is an one-time-mapping approach, just like CLIPS-OWL, allowing JRules production rules to use ontological knowledge. This is currently an ongoing work, without providing a detailed description about the transformation procedure.

SWRLJessTab [29] maps SWRL rules to Jess rules and OWL entailed facts from the Racer reasoner to simple triple-based Jess facts. Jess rules operate over Jess's KB and the inferred information is translated back to Racer in order to perform DL reasoning again. DL-Florid [25] is a combination of DL and F-Logic [48], where the latter is used as an ontology representation language only, following a frame-based syntax. The knowledge is finally translated into F-Logic facts (or XSB facts in F-OWL [49]). CLIPS-OWL is based totally on OO principles, using the OO model for physically representing ontology relationships and not triple-based facts.

Some examples of hybrid unidirectional approaches are the [50][26][22][51]. AL-log [50] is a combination of ALC DL and positive Datalog, where only concepts can only be used as constraints in rule bodies. A query is examined by using backward chaining and a DL reasoner in order to prove the DL constraints at runtime. DL-log [26] combines disjunctive Datalog with ALC, extending AL-log to property constraints in rule bodies. In [22], XSB is interfaced with Pellet and only DL constraints of the form $X : C$ are used in the rule bodies that are checked by the reasoner. In [51], a hybrid framework combines a rule language with a constraint language, following the typical hybrid architecture. These approaches utilize the DL reasoner "on the fly", without mapping the ontologies.

Note again that CLIPS-OWL has different application domains to the approaches that perform a *tight integration of rules and ontologies*, such as KAON2 [52], SWRL [53] and other homogeneous systems [54]. The goal of such approaches is to enable the efficient and decidable integration of rules and ontologies in order to express richer ontological relations. Our intention is not to manipulate ontologies with rules but to use the results of a DL reasoner in CLIPS-based applications in the form of COOL-based query models.

6. Conclusions and Future Work

There are different approaches that allow the already existing and well-known infrastructure of rule engines to gain access in the OWL ontological representation of information. We argue that there is not a single approach suitable for every problem, and the suitability depends on the application domain and user requirements.

In this paper, we presented CLIPS-OWL, a framework for transforming the results of OWL DL reasoners into the OO model that is supported by

CLIPS (COOL), allowing production rules to use the native infrastructure of the rule engine for answering queries related to OWL ontology instances. Our approach has been motivated by three main factors:

1. to allow the development of practical CLIPS rule programs able to query extensional OWL knowledge, based on the native CLIPS capabilities, preserving the existing architecture of CLIPS that has been proven highly stable and robust throughout the years of its use,
2. to exploit the ontology inferencing capabilities of the DL reasoning paradigm for handling OWL semantics, and
3. to achieve a more compact and efficient representation of ontology axioms in the rule engine through the use of OO principles, than having to explicitly store them in the form of facts, resulting in better query response performance in CLIPS.

CLIPS-OWL is implemented in Java using the Pellet DL reasoner and it is available as an open-source library. Currently, we are implementing our mapping algorithms in other OO rule engines, such as the forward-chaining inference rule engine of Drools[10] that allows the existence of a Java OO model (beans). CLIPS-OWL is being used in the domain of Software AntiPatterns [17], which represent software project management knowledge. We plan also to use CLIPS-OWL in the domain of Semantic Web Services in order to implement a knowledge-based Web service discovery framework using the capabilities of CLIPS.

References

[1] T.-P. Chang, R. Hull, Using witness generators to support bi-directional update between object-based databases (extended abstract), in: PODS '95: Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, ACM, 1995, pp. 196–207.

[2] A. Y. Levy, A. Rajaraman, J. J. Ordille, Querying heterogeneous information sources using source descriptions, in: VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996, pp. 251–262.

[10]`http://www.jboss.org/drools/`

[3] S. Bergamaschi, S. Castano, M. Vincini, D. Beneventano, Semantic integration of heterogeneous information sources, Data Knowl. Eng. 36 (3) (2001) 215–249.

[4] E. Simperl, Reusing ontologies on the semantic web: A feasibility study, Data & Knowledge Engineering 68 (10) (2009) 905 – 925.

[5] F. van Harmelen, D. L. McGuinness, OWL Web Ontology Language Overview, `http://www.w3.org/TR/2004/REC-owl-features-20040210/` (2004).

[6] B. Grau, I. Horrocks, B. Motik, B. Parsia, P. Patel-Schneider, U. Sattler, OWL 2: The next step for OWL, Web Semantics: Science, Services and Agents on the World Wide Web 6 (4) (2008) 309–322.

[7] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, 2003.

[8] L. Al-Jadir, C. Parent, S. Spaccapietra, Reasoning with large ontologies stored in relational databases: The ontomind approach, Data & Knowledge Engineering 69 (11) (2010) 1158 – 1180.

[9] J. Fang, Z. Huang, Reasoning with inconsistent ontologies, Tsinghua Science & Technology 15 (6) (2010) 687 – 691.

[10] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, Y. Katz, Pellet: A practical OWL-DL reasoner, Web Semantics: Science, Services and Agents on the World Wide Web 5 (2) (2007) 51–53.

[11] V. Haarslev, R. Möller, Racer: A Core Inference Engine for the Semantic Web, in: 2nd International Workshop on Evaluation of Ontology-based Tools (EON2003), CEUR-WS vol. 87, 2003, pp. 27–36.

[12] D. Tsarkov, I. Horrocks, FaCT++ Description Logic Reasoner: System description, in: Third International Joint Conference on Automated Reasoning (IJCAR), Vol. 4130 LNAI, 2006, pp. 292–297.

[13] F. Baader, U. Sattler, An Overview of Tableau Algorithms for Description Logics, Studia Logica 69 (2001) 5–40.

[14] R. Möller, V. Haarslev, Tableaux-based reasoning, in: S. Staab, R. Studer (Eds.), Handbook of Ontologies, Springer, 2009, pp. 509–528.

[15] G. Riley, CLIPS: An Expert System Building Tool, in: Proceedings of the Technology 2001 Conference, San Jose, CA, 1991.

[16] R. Rosati, On the finite controllability of conjunctive query answering in databases under open-world assumption, Journal of Computer and System Sciences 77 (3) (2011) 572 – 594.

[17] D. L. Settas, G. Meditskos, I. G. Stamelos, N. Bassiliades, Sparse: A symptom-based antipattern retrieval knowledge-based system using semantic web technologies, Expert Systems with Applications 38 (6) (2011) 7633 – 7646.

[18] E. J. Ruiz, B. C. Grau, I. Horrocks, R. Berlanga, Supporting concurrent ontology development: Framework, algorithms and tool, Data & Knowledge Engineering 70 (1) (2011) 146 – 164.

[19] J. de Bruijn, P. Bonnard, H. Citeau, S. Dehors, S. Heymans, R. Korf, J. Pührer, T. Eiter, State-of-the-art Survey of Issues, Tech. Rep. D3.1, EU-IST Integrated Project (IP) 2009-231875, ONTORULE (2009).

[20] G. Nalepa, W. Furmanska, Pellet-heart proposal of an architecture for ontology systems with rules, in: R. Dillmann, J. Beyerer, U. Hanebeck, T. Schultz (Eds.), KI 2010: Advances in Artificial Intelligence, Vol. 6359 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2010, pp. 143–150.

[21] G. Meditskos, N. Bassiliades, Rule-based OWL Reasoning Systems: Implementations, Strengths and Weaknesses, Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches, IGI Global, ISBN Number 978-1-60566-402-6, 2009.

[22] W. Drabent, J. Henriksson, J. Maluszynski, HD-rules: A Hybrid System Interfacing Prolog with DL-reasoners, in: Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services, Vol. 287, CEUR-WS, 2007.

[23] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, H. Tompits, Combining answer set programming with description logics for the Semantic Web, Artif. Intell. 172 (12-13) (2008) 1495–1539.

[24] R. Rosati, Semantic and Computational Advantages of the Safe Integration of Ontologies and Rules, in: Principles and Practice of Semantic Web Reasoning, Springer 3703, 2005, pp. 50–64.

[25] H. Kattenstroth, W. May, F. Schenk, Combining OWL with F-Logic Rules and Defaults, in: 2nd International Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services, Vol. 287, CEUR-WS, 2007, pp. 60–75.

[26] R. Rosati, DL+log: Tight Integration of Description Logics and Disjunctive Datalog, in: Tenth International Conference on Principles of Knowledge Representation and Reasoning, 2006, pp. 68–78.

[27] S. Bragaglia, F. Chesani, A. Ciampolini, P. Mello, M. Montali, D. Sottara, An hybrid architecture integrating forward rules with fuzzy ontological reasoning, in: HAIS (1), 2010, pp. 438–445.

[28] M. Grobe, Rdf, jena, sparql and the 'semantic web'., in: G. Farally-Semerad, K. J. McRitchie, E. Rugg (Eds.), SIGUCCS, ACM, 2009, pp. 131–138.

[29] C. Golbreich, A. Imai, Combining SWRL rules and OWL ontologies with Protege OWL Plugin, Jess, and Racer, in: 7th International Protege Conference, 2004.

[30] J. de Bruijn, R. Lara, A. Polleres, D. Fensel, OWL DL vs. OWL flight: conceptual modeling and reasoning for the semantic Web, in: Proceedings of the 14th international conference on World Wide Web, 2005, pp. 623–632.

[31] P. F. Patel-Schneider, I. Horrocks, A Comparison of Two Modelling Paradigms in the Semantic Web, Web Semantics: Science, Services and Agents on the World Wide Web, Elsevier 5 (4) (2007) 240–250.

[32] H. J. ter Horst, Completeness, Decidability and Complexity of Entailment for RDF Schema and a Semantic Extension Involving the OWL

Vocabulary, Web Semantics: Science, Services and Agents on the World Wide Web 3 (2-3) (2005) 79–115.

[33] G. Meditskos, N. Bassiliades, Combining a dl reasoner and a rule engine for improving entailment-based owl reasoning, in: International Semantic Web Conference, 2008, pp. 277–292.

[34] S. Heymans, J. D. bruijn, L. Predoiu, C. Feier, D. V. niewenborgh, Guarded hybrid knowledge bases, Theory Pract. Log. Program. 8 (3) (2008) 411–429.

[35] J. Mei, Z. Lin, H. Boley, $\mathcal{ALC}_p^u$: An Integration of Description Logic and General Rules, in: Web Reasoning and Rule Systems (RR), Spinger 4524, 2007, pp. 163–177.

[36] B. Grosof, I. Horrocks, R. Volz, S. Decker, Description Logic Programs: Combining Logic Programs with Description Logics, in: World Wide Web, ACM, Budapest, Hungary, 2003, pp. 48–57.

[37] G. Meditskos, N. Bassiliades, Dlejena: A practical forward-chaining owl 2 rl reasoner combining jena and pellet, Web Semantics: Science, Services and Agents on the World Wide Web 8 (1) (2010) 89 – 94.

[38] E. Kontopoulos, N. Bassiliades, G. Antoniou, Deploying defeasible logic rule bases for the semantic web, Data & Knowledge Engineering 66 (1) (2008) 116 – 146, including Special Section: Natural Language Processing and Information Systems (NLDB 2006) - Four selected and extended papers.

[39] L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, S. Liu, Towards a Complete OWL Ontology Benchmark, in: European Semantic Web Conference, Springer, 2006, pp. 125–139.

[40] A. Eberhart, Automatic Generation of Java/SQL Based Inference Engines from RDF Schema and RuleML, in: International Semantic Web Conference on The Semantic Web, Spinger, 2002, pp. 102–116.

[41] A. Kalyanpur, D. J. Pastor, S. Battle, J. A. Padget, Automatic Mapping of OWL Ontologies into Java, in: Sixteenth International Conference on Software Engineering and Knowledge Engineering, ISBN 1-891706-14-4, 2004, pp. 98–103.

[42] M. Völkel, Rdfreactor – from ontologies to programatic data access, in: Proc. of the Jena User Conference 2006, HP Bristol, 2006.

[43] D. Berardi, D. Calvanese, G. D. Giacomo, Reasoning on uml class diagrams, Artificial Intelligence 168 (1-2) (2005) 70 – 118.

[44] C. Puleston, B. Parsia, J. Cunningham, A. Rector, Integrating Object-Oriented and Ontological Representations: A Case Study in Java and OWL, in: International Semantic Web Conference (ISWC), Springer 5318, 2008, pp. 130–145.

[45] E. Oren, B. Heitmann, S. Decker, Activerdf: Embedding semantic web data into object-oriented languages, J. Web Sem. 6 (3) (2008) 191–202.

[46] S. Koide, H. Takeda, OWL-Full Reasoning from an Object Oriented Perspective, in: Asian Semantic Web Conference, 2006, pp. 263–277.

[47] G. Meditskos, N. Bassiliades, A Rule-Based Object-Oriented OWL Reasoner, IEEE Trans. on Knowl. and Data Eng. 20 (3) (2008) 397–410.

[48] M. Kifer, G. Lausen, J. Wu, Logical Foundations of Object-Oriented and Frame-Based Languages, Journal of ACM 42 (1995) 741–843.

[49] Y. Zou, T. W. Finin, H. Chen, F-OWL: An Inference Engine for Semantic Web, in: Formal Approaches to Agent-Based Systems, Springer 3228, 2004, pp. 238–248.

[50] F. M. Donini, M. Lenzerini, D. Nardi, A. Schaerf, Al-log: Integrating datalog and description logics, Journal of Intelligent Information Systems, Springer 10 (1998) 227–252.

[51] J. Henriksson, T. U. Dresden, Combining Safe Rules and Ontologies by Interfacing of Reasoners, in: Principles and Practice of Semantic Web Reasoning, 4th International Workshop, Budva, Montenegro, Revised Selected Papers, Springer, 2006, pp. 33–47.

[52] B. Motik, Kaon2 - scalable reasoning over ontologies with large data sets., ERCIM News 2008 (72).

[53] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, SWRL: A Semantic Web Rule Language Combining OWL

and RuleML, Tech. rep., W3C Member Submission (2004).
URL `http://www.w3.org/Submission/SWRL/`

[54] B. Motik, R. Rosati, Reconciling description logics and rules, J. ACM 57 (2008) 1–62.

A. Complete Mapping Example

In this section we present a mapping example that depicts every aspect of the transformation procedure. The ontology is depicted in Table 6 and refers to the university domain. More specifically, the following relationships are modeled:

$a1$: A chair is also a professor
$a2$: A person who is the head of a department is also a chair
$a3$: A person is a man or a woman
$a4$: All persons are individuals and vice versa
$a5$: A department is also an organization
$a6$: All organizations are institutes and vice versa
$a7$: The domain of the property `isHeadOf` is the concept `Individual`
$a8$: The domain of the property `isHeadOf` is the concept `Professor`
$a9$: The range of the property `isHeadOf` is the concept `Organization`
$a10$: The instance `csd` belongs to the `Department` concept
$a11$: The instance `nick` belongs to the `Person` concept
$a12$: `nick` is the head of the computer science department

A.1. Reasoning Results

After the DL reasoning procedure, we have the following $TBox_R$ results.

$$CN_R \equiv \{\texttt{owl:Thing}, \texttt{Chair}, \texttt{Professor}, \texttt{Person}, \texttt{Department}, \texttt{Man}, \texttt{Woman}, \\ \texttt{Individual}, \texttt{Organization}, \texttt{Institute}\}$$

$$H_R \equiv \{(\texttt{owl:Thing}, \texttt{owl:Thing}), (\texttt{Chair}, \texttt{Professor}), (\texttt{Chair}, \texttt{Person}), \\ (\texttt{Person}, \texttt{Individual}), (\texttt{Department}, \texttt{Organization}), \\ (\texttt{Man}, \texttt{Person}), (\texttt{Woman}, \texttt{Person}), (\texttt{Individual}, \texttt{Person}), \\ (\texttt{Organization}, \texttt{Institute}), (\texttt{Institute}, \texttt{Organization}) \\ (\texttt{Professor}, \texttt{owl:Thing})\}$$

Table 6: An ontology describing the university domain.

#	OWL Axioms (DL Syntax)
$a1$	$\texttt{Chair} \sqsubseteq \texttt{Professor}$
$a2$	$\texttt{Chair} \equiv \texttt{Person} \sqcap \exists\texttt{isHeadOf.Department}$
$a3$	$\texttt{Person} \equiv \texttt{Man} \sqcup \texttt{Woman}$
$a4$	$\texttt{Person} \equiv \texttt{Individual}$
$a5$	$\texttt{Department} \sqsubseteq \texttt{Organization}$
$a6$	$\texttt{Organization} \equiv \texttt{Institute}$
$a7$	$\top \sqsubseteq \forall\texttt{isHeadOf}^{-}\texttt{.Individual}$
$a8$	$\top \sqsubseteq \forall\texttt{isHeadOf}^{-}\texttt{.Professor}$
$a9$	$\top \sqsubseteq \forall\texttt{isHeadOf.Organization}$
$a10$	$\texttt{csd} : \texttt{Department}$
$a11$	$\texttt{nick} : \texttt{Person}$
$a12$	$\langle\texttt{nick,csd}\rangle : \texttt{isHeadOf}$

$$
\begin{aligned}
DOM_R \equiv \{&(\texttt{isHeadOf}, \texttt{Individual}), (\texttt{isHeadOf}, \texttt{Professor}),\\
&(\texttt{isHeadOf}, \texttt{Person})\}
\end{aligned}
$$

$$OBJ_R \equiv \{(\texttt{isHeadOf}, \texttt{Organization}), (\texttt{isHeadOf}, \texttt{Institute})\}$$

$$DAT_R \equiv \varnothing$$

Regarding the $ABox_R$ results, we have the following.

$$IN_R \equiv \{\texttt{csd}, \texttt{nick}\}$$

$$
\begin{aligned}
I_R \equiv \{&(\texttt{csd}, \texttt{Department}), (\texttt{csd}, \texttt{Organization}), (\texttt{csd}, \texttt{Institute}),\\
&(\texttt{nick}, \texttt{Person}), (\texttt{nick}, \texttt{Individual}), (\texttt{nick}, \texttt{Chair}), (\texttt{nick}, \texttt{Professor})\}
\end{aligned}
$$

$$ObjV_R \equiv \{(\texttt{nick}, \texttt{isHeadOf}, \texttt{csd})\}$$

$$DatV_R \equiv \varnothing$$

A.2. Defining the Delegators

In our example, there are two equivalent concept sets among named classes: {`Person`, `Individual`} and {`Organization`, `Institute`}. Assuming that the concepts Individual and Institute are selected as the delegators, respectively, the set DEL is defined as

$$\begin{aligned} DEL \equiv \{&(\texttt{Person}, \texttt{Individual}), (\texttt{Individual}, \texttt{Individual}), (\texttt{Organization},\\ &\texttt{Insitute}), (\texttt{Institute}, \texttt{Institute}), (\texttt{Professor}, \texttt{Professor}),\\ &(\texttt{Chair}, \texttt{Chair}), (\texttt{Man}, \texttt{Man}), (\texttt{Woman}, \texttt{Woman}), (\texttt{Department}, \texttt{Department}),\\ &(\texttt{owl:Thing}, \texttt{owl:Thing})\} \end{aligned}$$

In section 3.3.1 we described that all the non-delegator concepts should become superclasses of their delegator and subclasses of owl:Thing. Furthermore, every subclass relationship should refer only to delegators. Based on the set DEL, the H_D set is defined as

$$\begin{aligned} H_D \equiv \{&(\texttt{owl:Thing}, \texttt{owl:Thing}), (\texttt{Chair}, \texttt{Professor}), (\texttt{Chair}, \texttt{Individual}),\\ &(\texttt{Individual}, \texttt{Person}), (\texttt{Department}, \texttt{Institute}), (\texttt{Man}, \texttt{Individual}),\\ &(\texttt{Woman}, \texttt{Individual}), (\texttt{Institute}, \texttt{Organization}), (\texttt{Professor},\\ &\texttt{owl:Thing}), (\texttt{Person}, \texttt{owl:Thing}), (\texttt{Organization}, \texttt{owl:Thing})\} \end{aligned}$$

Furthermore, the domain restrictions of properties and the range restrictions of object properties should refer now only to delegators. The sets DOM_D and OBJ_D are defined as follows.

$$DOM_D \equiv \{(\texttt{isHeadOf}, \texttt{Individual}), (\texttt{isHeadOf}, \texttt{Professor})\}$$

$$OBJ_D \equiv \{(\texttt{isHeadOf}, \texttt{Institute})\}$$

A.3. Multiple Domain/Range Restrictions

The isHeadOf property has two domain restrictions and therefore

$$DOM_{MC} \equiv \{(\texttt{isHeadOf}), \{\texttt{Individual}, \texttt{Professor}\}\}$$

Since the property has multiple domain restrictions, a system concept should be generated. The Individual and Professor concepts are not hierarchically related, therefore, the *classGen* function would generate a system concept, e.g. IndividualProfessor, as subclass of the two concepts that will be added to the CN_M and H_M sets. In that way, the DOM_M, OBJ_M, CN_M and H_M sets are defined as follows.

$$DOM_M \equiv \{(\texttt{isHeadOf}, \texttt{IndividualProfessor})\}$$

$$OBJ_M \equiv \{(\texttt{isHeadOf}, \texttt{Institute})\}$$

$$CN_M \equiv CN_D \cup \{\texttt{IndividualProfessor}\}$$

$$H_M \equiv H_D \cup \{(\texttt{IndividualProfessor}, \texttt{Individual}), (\texttt{IndividualProfessor}, \texttt{Professor})\}$$

A.4. Instance Management

The instance concept memberships should refer only to delegator concepts. Therefore, the I_D set is defined as

$$I_D \equiv \{(\texttt{csd}, \texttt{Department}), (\texttt{csd}, \texttt{Institute}), (\texttt{nick}, \texttt{Individual}), (\texttt{nick}, \texttt{Chair}), (\texttt{nick}, \texttt{Professor})\}$$

In order to handle the multiple instance concept memberships, the set I_{MC} is defined as

$$I_{MC} \equiv \{(\texttt{csd}, \{\texttt{Department}, \texttt{Institute}\}), (\texttt{nick}, \{\texttt{Individual}, \texttt{Chair}, \texttt{Professor}\})\}$$

The `csd` instance belongs to the `Department` and `Institute` concepts. However, `Department` is subclass of `Institute` in H_M, and therefore, only the `Department` concept will be considered as the single concept membership in I_M, without generating a new concept. Similarly, `Chair` is subclass of `Individual` and `Professor`, and therefore, there is no need to generate a system concept. The I_M set is defined as

$$I_M \equiv \{(\texttt{csd}, \texttt{Department}), (\texttt{nick}, \texttt{Chair})\}$$

A.5. Mapping Phase

Based on the CN_M, H_M, DOM_M, OBJ_M and $DAT_M \equiv \varnothing$ sets, the $COOL_{sch}$ set is defined as

$$\begin{aligned}
COOL_{sch} \equiv \{&\langle \texttt{owl:Thing}, \{\texttt{owl:Thing}\}, \varnothing\rangle,\\
&\langle \texttt{Chair}, \{\texttt{IndividualProfessor}\}, \varnothing\rangle,\\
&\langle \texttt{Individual}, \{\texttt{Person}\}, \varnothing\rangle,\\
&\langle \texttt{Department}, \{\texttt{Institute}\}, \varnothing\rangle,\\
&\langle \texttt{Man}, \{\texttt{Individual}\}, \varnothing\rangle,\\
&\langle \texttt{Woman}, \{\texttt{Individual}\}, \varnothing\rangle,\\
&\langle \texttt{Institute}, \{\texttt{Organization}\}, \varnothing\rangle\\
&\langle \texttt{Professor}, \{\texttt{owl:Thing}\}, \varnothing\rangle\\
&\langle \texttt{Organization}, \{\texttt{owl:Thing}\}, \varnothing\rangle\\
&\langle \texttt{Person}, \{\texttt{owl:Thing}\}, \varnothing\rangle\\
&\langle \texttt{IndividualProfessor}, \{\texttt{Individual, Professor}\},\\
&\quad \langle \texttt{isHeadOf}, \texttt{Institute}\rangle\}\rangle\\
&\}
\end{aligned}$$

Based on the IN_M,I_M, $ObjV_M$ and $DatV_M \equiv \varnothing$ sets, the $COOL_{data}$ set is defined as

$$COOL_{data} \equiv \{\langle \texttt{csd}, \texttt{Department}, \varnothing\rangle, \langle \texttt{nick}, \texttt{Chair}, \{\langle \texttt{isHeadOf}, \texttt{csd}\rangle\}\rangle\}$$

Based on the *codeGenerator* procedure and the $COOL_{sch}$, $COOL_{data}$ sets, the generated COOL code is the following.

```
(defclass owl:Thing (is-a USER))
(defclass Person (is-a owl:Thing))
(defclass Professor (is-a owl:Thing))
(defclass Organization (is-a owl:Thing))
(defclass Individual (is-a Person))
(defclass Institute (is-a Organization))
(defclass Man (is-a Individual))
(defclass Woman (is-a Individual))
(defclass IndividualProfessor (is-a Individual Professor)
   (multislot isHeadOf
      (type INSTANCE-NAME)(allowed-classes Institute)))
(defclass Chair (is-a IndividualProfessor))
(defclass Department (is-a Institute))
```

```
(make-instance [csd] of Department)
(make-instance [nick] of Chair (isHeadOf [csd]))
```

B. Proofs

In this section we present the proofs for the correctness of our transformation procedure, regarding the assumption that the ABox reasoning results, that is, the instance concept memberships and instance property values, are preserved in the generated COOL model.

B.1. Instance Concept Memberships

We will prove that the instance concept memberships of an ontology are preserved in the generated COOL model, that is, if an instance belongs to a set of concepts in the ontology, then the corresponding object belongs to the corresponding classes in COOL. More formally, we will prove that $\forall i, c\ (i, c) \in H_R \vdash \exists i', c' \langle i', c^{i'}, PV^{i'} \rangle \in COOL_{data} \wedge \langle c', S^{c'}, P^{c'} \rangle \in COOL_{sch} \wedge i' = f_{COOL_{data}}(f_{M_{data}}(f_{D_{data}}(i))) \wedge c' = f_{COOL_{sch}}(f_{M_{sch}}(f_{D_{sch}}(c))) \wedge i' \in EXT(c')$, where $EXT(c)$ is the set of all the objects (direct and indirect) of class C in COOL.

B.1.1. Single Instance Concept Membership

Let an instance i belong to a single ontology concept c in the reasoning results, that is, $i \in IN_R \wedge c \in CN_R \wedge (i, c) \in I_R \wedge \nexists c'(c' \neq c \wedge (i, c') \in I_R))$. Since $c \in CN_R \wedge CN_R = CN_D \wedge CN_D \subseteq CN_M \vdash c \in CN_M$. According to the definition of $COOL_{sch}$, we have that $\forall c \in CN_M, \exists c_o \in COOL_{sch} : c_o = \langle c, S^c, P^c \rangle$. Therefore, an ontology concept in CN_R is mapped on a single class in COOL that is represented by c_o. Note that there might be classes in COOL that are not mapped by any ontology concept, due to the system generated concepts that are generated by the transformation procedure.

Regarding the instance, we have that $i \in IN_R \wedge IN_R = IN_D \wedge IN_D = IN_M \vdash i \in IN_M$. According to the definition of $COOL_{data}$, we have that $\forall i \in IN_M, \exists i_o \in COOL_{data} : i_o = \langle i, c'^i, PV^i \rangle \wedge (i, c'^i) \in I_M$ and therefore, $c'^i \in CN_M$, based on the definition of CN_M. Furthermore, $COOL_{sch}$ is defined in such a way, so as $\forall c'^i \in CN_M, \exists c'^i_o \in COOL_{sch} : c'^i_o = \langle c'^i, S^{c'^i}, P^{c'^i} \rangle$. Therefore, the ontology instance i is mapped on a single object i_o of the class c'^i_o in COOL.

In section 3.3.1 we analyzed the procedure of assigning delegators. Each concept has a delegator, that is, $\forall c, \exists a(c, a) \in DEL$. Furthermore, we explained that every object is forced to belong to delegator concepts in the I_D

set. Therefore, $(i, c) \in I_R \wedge (c, a) \in DEL \vdash (i, a) \in I_D$. Since i is assumed to belong to a single concept, then $(i, \{a\}) \in I_{MC}$ and $(i, a) \in I_M$. However, since $(i, a) \in I_M$, $(i, c'^i) \in I_M$ and i is an instance of a single concept in I_M, then we can conclude that $a = c'^i$ and $a_o = c'^i_o$. Therefore, i_o is an object of the class a_o in COOL. In order to prove that i_o is an object of the class c_o, we should prove that a_o is a direct or indirect subclass of c_o.

From the definition of the set H_D, we have that $(c, a) \in DEL \vdash (a, c) \in H_D$. The set H_M defines that $H_D \subseteq H_M$ and therefore, $(a, c) \in H_M$. Therefore, if $a_o = \langle a, S^a, P^a \rangle$, then $c \in S^a$ in $COOL_{sch}$, which means that a_o is a subclass of c_o in COOL. Therefore, i_o is an object of the class c_o in COOL, in accordance to the initial assumption that i is an instance of the concept c in the ontology.

B.1.2. Multiple Instance Concept Memberships

Let an instance i belong to multiple ontology concepts in the reasoning results, that is, $\exists i \exists c^1 \exists c^2 ... \exists c^N (i \in IN_R \wedge c^1 \in CN_R \wedge c^2 \in CN_R \wedge ... \wedge c^N \in CN_R \wedge (i, c^1) \in I_R \wedge (i, c^2) \in I_R \wedge ... \wedge (i, c^N) \in I_R)$, $N > 1$. It holds that $\forall k, c^k \in CN_R \wedge CN_R = CN_D \wedge CN_D \subseteq CN_M \vdash c^k \in CN_M$. Based on the definition of $COOL_{sch}$, we have that $\forall c^k \in CN_M, \exists c^k_o \in COOL_{sch} : c^k_o = \langle c^k, S^{ck}, P^{ck} \rangle$. Therefore, the concept c^k is mapped on a single class c^k_o in COOL.

Regarding the instance, we have that $i \in IN_R \wedge IN_R = IN_D \wedge IN_D = IN_M \vdash i \in IN_M$. According to the definition of $COOL_{data}$ we have that $\forall i \in IN_M, \exists i_o \in I_o : i_o = \langle i, c'^i, PV^i \rangle \wedge (i, c'^i) \in I_M$ and, if $(i, c'^i) \in I_M$, then $c'^i \in CN_M$, based on the definition of CN_M. Furthermore, the set $COOL_{sch}$ is defined in such a way, so as $\forall c'^i \in CN_M, \exists c'^i_o \in COOL_{sch} : c'^i_o = \langle c'^i, S^{c'^i}, P^{c'^i} \rangle$. Therefore the ontology instance i is mapped on a single object i_o of the class c'^i_o in COOL.

Furthermore, based on the definition of the set I_D, we have that $\forall k (i, c^k) \in I_R \wedge (c^k, a^k) \in DEL \vdash (i, a^k) \in I_D$, since $\forall k, \exists a^k (c^k, a^k) \in DEL$. Since i belongs to multiple concepts, we have that $(i, \{a^1, a^2, ..., a^N\}) \in I_{MC}$ and $(i, classGen(a^1, a^2, ..., a^N)) \in I_M$. Let $a^i = classGen(a^1, a^2, ..., a^N)$, that is, $(i, a^i) \in I_M$. According to the definition of H_M, it holds that $\forall k (a^i, a^k) \in H_M$. Since $(i, a^i) \in I_M$, $(i, c'^i) \in I_M$ and i is an instance of a single concept in I_M, we conclude that $a^i = c'^i$ and therefore, $a^i_o = c'^i_o$. Thus, i_o is an object of the class a^i_o in COOL. In order to prove that i_o is an object of all the a^k_o classes, we should prove that a^i_o is a direct or indirect subclass of that classes in COOL.

According to the definition of H_D, we have that $\forall k(c^k, a^k) \in DEL \vdash (a^k, c^k) \in H_D$. However, $H_D \subseteq H_M$ and therefore, $\forall k(a^k, c^k) \in H_M$. Furthermore, we have shown previously that $\forall k(a^i, a^k) \in H_M$. Therefore, it also holds that $\forall k(a^i, c^k) \in H_M$ (transitive relationship). Thus, if $a^i_o = \langle a^i, S^{a^i}, P^{a^i} \rangle$, we have from the set $COOL_{sch}$ that $\forall k, c^k \in S$, which means that $\forall k, a^i_o \ll_o c^k_o$. Therefore, the class a^i_o is subclass of all the c^k_o classes in COOL and i_o is an object of all these classes, due to the class subsumption relationships in the OO environment of COOL.

B.2. Instance Property Values

We will prove that the property values of an ontology instance are preserved in the COOL model, that is, a query for the slot values of an object in COOL returns the same values that the corresponding instance has in the ontology. More formally, we will prove that $\forall x, p, y(x, p, y) \in ObjV_R \cup DatV_R \vdash \exists c^x, PV^x \langle x, c^x, PV^x \rangle \in COOL_{data} \wedge \langle p, y \rangle \in PV^x$.

The transformation procedure does not modify the property-value sets. Therefore, $ObjV_M = ObjV_D = ObjV_R$, $DatV_M = DatV_D = DatV_R$ and $(x, p, y) \in ObjV_R \cup DatV_R \rightarrow (x, p, y) \in ObjV_M \cup DatV_M$. Moreover, $x \in IN_R \wedge IN_R = IN_D \wedge IN_D = IN_M \vdash x \in IN_M$. According to the definition of $COOL_{data}$, $\forall x \in IN_M, \exists i^x_o \in COOL_{data} : i^x_o = \langle x, c^x, PV^x \rangle \wedge (x, c^x) \in I_M$. The set PV^i is defined as $PV^i = \{\langle p, v \rangle \mid (i, p, v) \in ObjV_M \cup DatV_m\}$. Therefore, $\exists c^x, PV^x \langle c, c^x, PV^x \rangle \in COOL_{data} \wedge \langle p, y \rangle \in PV^x$.